CS 214, Winter 2022 BFS Project

Due: Mon 02/21, 2022

Name: Matthew Duc Nguyen
Student ID #: 862254913
Github ID: matthewduc

BFS Project Mid Report Progress

I am reviewing previous lectures of all functions to implement for this project.
Homework 0 introduced the reduce function with granularity control. With this method of implementation, I'm attempting to use with scan, pack, and flatten to find prioritize run times. Reviewing granularity control with reduce:

```
size_t reduce(size_t* A, int n, size_t thresh) {
    if (n < thresh) {
        size_t ans = 0;
        for (size_t i = 0; i<n; i++)
            ans += A[i];
        return ans;
    }
size_t L, R;
L = cilk_spawn reduce(A, n/2, thresh);
R = reduce(A+n/2, n-n/2, thresh);
cilk_sync;
return L+R;
}
```

Through the use of a threshold, the overhead of creating a new parallel task and joining them is controlled when the number of calculations in the task is small (computed sequentially). Looking at the solution for Homework 1 with the integral function:

```
template <class Func>
double integral(const Func& f, size_t n, double low, double high) {
    double ans = 0.0;
    double delta = (high-low)/(n+0.0);
    auto g = [&] (size_t i) -> double {
    double x = low + delta*i;
    return f(x);
};
// g is given as a function.
// g(i) is the i-th element to use in reduce
reduce(0, n, g, ans);
return ans*delta;
}
```

The integral function introduces math functions and a different version of reduce where the i-th element is not stored.
The following is reduce where the i-th element is not stored and just called to retrieve value in the base case:

```
size_t reduce(int start, int n, function f) {
    if (n==1) return f(start);
    int L, R;
    L = cilk_spawn reduce(start, n/2, f);
    R = reduce(start+n/2, n-n/2, f);
    cilk_sync;
    return L+R;
}
```

I am receiving errors running this version of reduce on XE-01.
Scan functions from lecture maintains a left sum array to store offset in computing prefix sum:

```
B[] = scan(A[], n) {
    LS = array[n-1];
    B = array[n];
    scan_up(A, LS, 0, n);
    scan_down(B, LS, 0, n, 0);
}
scan_up(A[], LS[], n) {
    if (n==1) return A[0];
    m = n/2;
    cilk_spawn l = scan_up(A, LS, m);
    r = scan_up(A+m, L+m, n-m);
    cilk_sync;
    LS[m-1] = 1; // store the "leftsum"
    return l+r;
}
scan_down(A[], B[], LS[], n, offset) {
    if(n==1) {
        B[0] = offset + A[0];
        return;
    }
    m = n/2;
    cilk_spawn scan_down(B,LS,m,LS[m];
    scan_down(B+m, LS+m, n-m, LS[m-1];
}
```

Using mathematical functions introduced in the integral function and scan, filter or packing outputs an array that satisfy the function using a flag array as follows:

```
Filter(A, n, B, f) {
    new array flag[n], ps[n];
    cilk_for (int i = 1; i<= n; i++){
        flag[i] = f(A[i]);
    }
    ps = scan(flag,n);
    cilk_for(int i = 1; i<= n; i++){
        if f(A[i]){
            B[ps[i]] = A[i];
        }
    }
}
```

From Breathe First Search graph 1 and 2 lectures, BFS visits vertices in order of distance from s. In parallel, atomic CAS ensures that each node in a vertex's neighbor is visited once:

```
x is in the current frontier
for y in x's neighbors {
    if((!visited[y] && CAS(&visited[y], false, true))
        let x visit y
}
```

Get effective neighbors:

```
x is in the current frontier
flag[0...deg(x)] = {false};
for i = 0 to deg(x) {
    y = CSR[offset[x]+i];
    if ((!visited[y]) && CAS(&visited[y], false, true))
        flag[i] = true;
}
Pack/Filter x's neightbors based on flag[i]
```

Flatten algorithm takes a nested sequence of 1-D arrays (effective neighbors) and outputs into one 1-D array:

```
Flatten(A,n) {
    cilk_for(int i=0; i<=n;i++){
        S[i] = |A[i]|;
    }
    offset = scan_exclusive(S,n);
    cilk_for(int i=0; i<=n;i++){
        off = offset[i];
        cilk_for((int j=0; j<=S[i];j++){
            B[off+j]=A[i][j];
        }
    }
    return B;
}
```

From lecture, generating next frontier is as follows:

```
// shared Boolean array visited[]
// visited[v] = true means v has been visited
next_frontier(G, frontier){
    cilk_for(int k=0; k<=|frontier|, k++){
        x = frontier[k];
        nghs = G[x].out_nghs;
        bool flag[0...|nghs|] = {false};
        // visit deg[x] neighbors of x
        for (int i =0; i<=deg(x);i++){
            y = nghs[i]; // y is the i-th neighbor
            if ((!visited[y]) && CAS(&visited[y], false, true)) // effective neighbor
            flag[i] = true;
        }
        Pack/Filter x's neighbors based on flag[i];
        effective_nghs[k] = filter(nghs, flag);
    }
    return flatten(effective_nghs);
}
```

Parallel BFS

```
BFS(G(V,E), s) {
    frontier = {s};
    n = |V|;
    visited = array(n, false);
    visited[s] = true;
    parent = array(n, -1);
    while(|frontier|) {
        frontier = next_frontier(G, frontier);
    }
    return parent;
}
```

Dense mode for BFS given G(V,E):

```
cilk_for each node in V
    skips visited nodes, otherwise, use v to visit all its neighbors
```

```
bool CAS(bool* p, bool vold, bool vnew) {
    if (*p==vold){
        *p = vnew;
        return true;
    }
    else false;
}
```

```
bool* visited = new bool[n];
cilk_for (int i = 0; i < n; i++) visited[i] = false;

 for (int i = offset[cur_node]; i < offset[cur_node+1]; i++) {
    int v = E[i];
    if((!visited[v]) && CAS(&visited[v], false,true)) {
        if (dist[v] == -1) {
            dist[v] = cur_dis + 1;
            q[end++] = v;
        }
        visited[v] = true;
    }
}
```

I am in the process of implementing functions and dense mode (above), and pushing to the github repo. I am also evaluating the best between cilk and non-cilk versions prioritizing for project runtimes.
Most up-to-date code constantly pushed to Github: project2-matthewduc